Štetje z zanko `for`

V programih pogosto potrebujemo še dva tipa zank. Pogosto bomo želeli nekaj ponoviti vnaprej predpisano (ali pred zanko izračunano) število ponovitev, recimo petkrat ali desetkrat. Prav tako bomo pogosto želeli, da program "šteje" od 10 do 20, od 0 do 100 ali kaj takega.

Za oboje bo poskrbela kar zanka for, ob pomoči priročne funkcije `range`.

Funkcija `range` je na prvi pogled povsem neimpresivna. Vrača sezname zaporednih števil.

V Pythonu 3.0 je ta funkcija narejena boljše kot v različicah pred njim, vendar je nova oblika za začetnika manj očitna, zato bomo za trenutek skočili v stari Python 2.7. Kar sledi, v novem Pythonu namreč ne bi dalo enakih izpisov - imelo pa bi (praktično) ekvivalenten učinek.

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(10, 20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> range(16)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Kaj dela, je očitno. In očitno ... čudno. `range(a, b)` vrne seznam celih števil od `a` do `b`, vključno s prvim, a brez zadnjega. Če podamo le en argument, pa začne z ničlo. Zakaj vključi prvega, zadnjega pa ne? Mogoče je bil tisti, ki je to programiral, malo šlampast? Ni dobro razmislil? Ali pa je preveč razmišljal in se ni mogel odločiti, zato se je odločil za čuden kompromis?

Niti ne. Funkcija `range` je narejena točno tako, kot je treba. Najprej: če si na morju od 17 do 22 julija, koliko dni si bil na morju? Kadar se znajdem pred takšno uganko (in pred njo se znajdem vsakič, ko moram fakulteti napisati poročilo s službene poti), štejem na prste. Ja, vem, 22 - 17 + 1, ampak ... zihr je zihr.

V Pythonu je to preprosteje. Koliko elementov ima seznam, ki ga vrne `range(17, 22)`? 22 - 17 = 5. Zato, ker seznam vsebuje prvi element, ne pa tudi zadnjega. (Temu lahko, z vidika počitnic, rečemo hotelski izračun: spat si šel 17, 18, 19, 20 in 21, ne pa tudi 22. Pet noči.

Koliko elementov ima seznam, ki ga vrne `range(7)`? Sedem. Zato ker vsebuje prvi element in ne zadnjega, in ker se začne z 0.

Funkcija `range` je torej narejena praktično, ker se težko zmotimo glede tega, koliko elementov bo vrnila.

Lepo se tudi sešteva. `range(10, 15) + range(15, 20)` so ravno vsa števila od 10 do 20. Če bi bil `range(10, 15)` enak `[10, 11, 12, 13, 14, 15]`, bi se v

`range(10, 15) + range(15, 20)` število 15 ponovilo dvakrat.

Vem, da vas (najbrž) nisem prepričal. Da je res dobro, da je `range` tako čuden, boste zares razumeli ob nečem v zvezi s seznami. In še to šele takrat, ko vam zlezejo bolj pod kožo. Do takrat se zanesimo na argument avtoritete: to je dobro zato, ker je Demšar tako rekel.

Funkcija ima lahko še en, tretji argument. Ta predstavlja korak.

```
>>> range(5, 15, 2)
[5, 7, 9, 11, 13]
```

Korak je lahko tudi negativen.

```
>>> range(10, 0, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> range(10, 0, -3)
[10, 7, 4, 1]
>>> range(10, 0, -5)
[10, 5]
```

Funkcijo `range` uporabljamo skoraj izključno v zankah `for`.

V prvi zanki `while`, ki smo jo napisali, smo izpisali števila od 1 do 10, takole nekako:

```
i = 1
while i < 11:
    print(i)
    i += 1
```

Z zanko `for` je preprostejši.

```
for i in range(1, 11):
    print(i)
```

Seveda ostaja odprto vprašanje, zakaj bi si kdo tako želel izpisati števila od 1 do 10. Štetje je kljub temu uporabno tudi za kaj drugega, ne le za izpisovanje.

Praštevila

Napisati želimo program, ki pove, ali je dano število praštevilo. Za to bomo preverili vsa števila od `2` do `n - 1`: če katerokoli od njih deli `n`, potem `n` ni praštevilo.

```
n = int(input("Vpiši število"))

je_prastevilo = True
for i in range(2, n):
    if n % i == 0:
        je_prastevilo = False
```

V začetku predpostavimo, da je število praštevilo (`je_prastevilo = True`). Če nato med števili med 2 in n (`for i in range(2, n)`) naletimo na njegov delitelj (`if n % i == 0`), pa presodimo, da število pač ne more biti praštevilo (`je_prastevilo = False`). Mimogrede naj opozorimo na napako, na katero stalno opozarjamo:

```
n = int(input("Vpiši število"))

je_prastevilo = True
for i in range(2, n):
    if n % i == 0:
        je_prastevilo = False
    else:
        je_prastevilo = True
```

Ko je število enkrat obsojeno kot sestavljeno, ga ne moremo več "pomilostiti" nazaj v praštevilo. Če je tako, pa tudi ni potrebe, da bi po tem, ko enkrat odkrijemo delitelj, sploh še preverjali naslednje potencialne delitelje. Zdaj že vemo: `break`.

```
n = int(input("Vpiši število"))

je_prastevilo = True
for i in range(2, n):
    if n % i == 0:
        je_prastevilo = False
        break
```

Indeksiranje

Do elementov seznamov, terk in nizov ne pridemo le z zanko for. Dobimo jih lahko tudi tako, da preprosto zahtevamo element na tem in tem mestu. Vzemimo za primer seznam imen. Spomnimo se, kako je videti.

```
>>> imena = ['Anze', 'Benjamin', 'Cilka', 'Dani', 'Eva', 'Franc']
```

Spremenljivka `imena` vsebuje šest elementov. Če hočemo dostopati do posameznega elementa - recimo izpisati njegovo vrednost - povemo njegov indeks, "zaporedno številko". Zapišemo jo v oglate oklepaje za imenom spremenljivke, takole:

```
>>> imena[2]
'Cilka'
```

Se pravi, `imena[2]` vrne drugi element seznama.

Emmm, drugi?! Mar ni drugi element Benjamin, ne Cilka? Drži, ampak kakor šteje python, je drugi element Cilka. Anže je pa ničti. Ne le v pythonu, skoraj v vseh jezikih štejemo od 0. Prvi element ima *indeks* 0, drugi element 1 in tretji 2. Zakaj? Razlogi so tehnični in praktični. Tehničnega boste razumeli, če boste

kdaj programirali v C-ju. Tradicionalno indeks pomeni "odmik od začetka": ko je odmik 0, dobimo prvi element in ko je odmik 2, se odmaknemo dva elementa, torej pristanemo na tretjem (torej: tistem, ki mu normalni ljudje rečejo tretji). Medtem ko je v Cju (in še nižjih jezikih) ta, tehnični, argument morda smiseln, upravičimo štetje od ničle v Pythonu in drugih višjih jezikih s praktičnostjo: reči se na ta način lepše izidejo. Boste videli.

Če je koga zmedlo, povejmo na glas: oglati oklepaji imajo dve vlogi. Prej smo jih uporabljali, da smo vanje zaprli seznam, zdaj vanje zapiramo indekse. Naj vas to ne vznemirja, python bo že pravilno razumel, kaj mislite, celo v tako hecnih situacijah, kot je tale:

```
>>> [3, 1, 4, 1, 5, 9][2]
4
```

Prvi oklepaji definirajo seznam, drugi zaprejo indeks, 2, ki pove, kateri element tega seznama nas zanima. [Taka raba je nenavadna, ni pa neuporabna. Kako se vam zdi tole: `["moski", "zenska"][spol]`, kjer je `spol` enak `True`, če gre za žensko in `False`, če za moškega? In tole: `"MŽ"[spol]`?

Tako kot sezname indeksiramo tudi terke in nize.

```
>>> 'Benjamin'[0]
'B'
>>> 'Benjamin'[2]
'n'
```

Kadar seznam vsebuje sezname, bomo včasih uporabljali dvojne indekse.

V neki dodatni domači nalogi smo sestavili spiralo iz 10000 števil. Mislim, da noben študent ni bil tako zagnan, da bi res sestavil celotno spiralo in jo zapisal v kak seznam seznamov. Lahko pa si privoščimo zapisati spiralo iz primera.

```
spirala = [
    [37, 36, 35, 34, 33, 32, 31],
    [38, 17, 16, 15, 14, 13, 30],
    [39, 18,  5,  4,  3, 12, 29],
    [40, 19,  6,  1,  2, 11, 28],
    [41, 20,  7,  8,  9, 10, 27],
    [42, 21, 22, 23, 24, 25, 26],
    [43, 44, 45, 46, 47, 48, 49]
]
```

Do tretje vrstice pridemo tako.

```
>>> spirala[3]
[40, 19,  6,  1,  2, 11, 28]
```

In drugi element iz tretje vrstice je potem

```
>>> spirala[3][2]
6
```

Ni razloga, da bi se bali dvojnih ali celo trojnih indeksov. Samo potem postane program nekoliko nepregleden. (V kost za glodanje C-jašem pa naj vprašam, ali `"Benjamin"[3][0][0][0][0]` kaj izpiše ali se pritoži, da je tu nekaj narobe. Ampak to je pa *res* neuporabno.)

Kaj se zgodi, če je indeks prevelik? Nič lepega.

```
>>> "Benjamin"[100]
Traceback (most recent call last): File "",
line 1, in IndexError: string index out of range
```

Kako velik pa je lahko indeks? Če ima niz osem črk in je prva ničta, je zadnja sedma. Indeks mora biti torej manjši od dolžine - največji dovoljeni indeks je tisto, kar vrne funkcija `len`, minus 1.

Python (in še marsikateri današnji jezik) ima še en trik: indeksiranje s konca: -1 je zadnji element, -2 predzadnji in tako naprej. (Če koga srbi, da bi vprašal, čemu ni zadnji element -0, naj se ugrizne v jezik, ali pa ga zatožim fakultetnim matematikom!)

```
>>> 'Benjamin'[-1]
'n'
>>> 'Benjamin'[-2]
'i'
>>> 'Benjamin'[-3]
'm'
```

V domačih nalogah in na izpitih pogosto videvam reči, kot je `s[len(s) - 5]`. To je pohvalno, saj pomeni, da zna študent programirati tudi v drugih jezikih, kjer bi moral pisati `s[strlen(s) - 5]` (C in podobni) ali, recimo, `s[count(s) - 5]` (php). V pythonu naj se pa kar lepo navadi, da lahko napiše `s[-5]`.

Rezanje

Poleg indeksiranja, ki vrača elemente nizov, seznamov, terk (in še česa), pozna Python še rezanje (*slice*), ki vrača dele nizov, seznamov, terk (in še česa). Rezino opišemo z indeksom prvega elementa in indeksom prvega elementa, ki ga ne želimo več vključiti v rezino. Med indeksa postavimo dvopičje. Se pravi, rezina 2:5 pomeni vse elemente od onega z indeksom 2 do tistega z indeksom 4 (ne 5!).

Smo to že kje videli? Smo, seveda. Funckija `range` uporablja natančno isto logiko. Tako kot pri `range` je tudi pri rezanju to odlična ideja.

Če študentom od drugega do petega, ki sedijo v prvi vrsti, naj vstanejo: koliko študentov bo vstalo? V običajnem svetu štirje, namreč drugi, tretji, četrti in peti. Štirje, čeprav je 5 - 2 pravzaprav enako 3. V svetu Pythonovih rezin vstanejo samo trije, namreč drugi, tretji in četrti (oziroma tretji, četrti in peti, če štejemo od ena, ne od nič).

Še en primer - kasneje bomo videli, zakaj je priročen: kaj, če naročim, naj

vstanejo vsi od drugega do petega, čez pol minute pa še vsi od petega do osmega? Koliko jih stoji? In predvsem, kolikokrat stoji peti? Dvakrat? Na prstih? V Pythonu je vstal samo drugič: prvič so vstali študenti 2, 3, in 4, drugič pa 5, 6 in 7. Skupaj jih stoji 5-2=3 iz prve runde in 8-5=3 iz druge. Skupaj šest.

Sicer pa smo prav tadva primera obdelovali že pri `range`.

Oglejmo si, kaj na to pravi Benjamin.

```
>>> b[2:5]
'nja'
>>> b[5:8]
'min'
>>> b[2:5]+b[5:8]
'njamin'
```

Spodnjo ali zgornjo mejo smemo tudi izpustiti. V tem primeru dobimo vse elemente od začetka oz. do konca. Tu bosta izkazali svojo moč prav obe navidez neintuitivni pravili - štetje od 0 in to, da rezina ne vključuje zadnjega elementa.

```
>>> b[:5]
'Benja'
>>> b[5:]
'min'
```

`b[:5]` vrne prvih pet elementov. `b[5:]` vrne vse od petega naprej; ker štejemo od 0, to pomeni, da izpustimo prvih pet. Se pravi, če želimo nizu `s` odbiti prvih pet znakov, bomo rekli

```
s = s[5:]
```

Da peti študent vstane le enkrat, nam pride prav, če želimo v niz kaj vriniti. Če bi radi za petim znakom niza vrinili X, to storimo takole:

```
>>> b[:5] + "X" + b[5:]
'BenjaXmin'
```

Ste opazili, da nam sploh ni bilo treba pomisliti na to, da štejemo od 0? Vidite, kako naravno je to. Če bi šteli od 1, bi bilo tole precej bolj zapleteno.

Še dodatne možnosti prinese indeksiranje od zadaj. Iz niza lahko poberemo, recimo, elemente od predpredpredzadnjega (-5) do predzadnjega (-2). Koliko jih bo? Trije, seveda.

```
>>> b[-5:-2]
'jam'
```

No, tegale najbrž ne uporabimo velikokrat. Pač pa nas pogosto zanimajo, recimo, zadnji štirje znaki. Ali pa vsi razen zadnjih štirih.

```
>>> film = "Babylon 5 - 3x04 - Passing through Gethsemane.avi"
>>> film[-4:]
'.avi'
```

```
>>> film[:-4]
'Babylon 5 - 3x04 - Passing through Gethsemane'
```

Mimogrede, tole sicer varneje delamo s funkcijo, ki smo jo že omenili, namreč `splitext`, ki prejme kot argument ime datoteke in vrne terko z dvema elementoma, imenom datoteke brez končnice in končnico.

Kako pa bi od niza odbili prve tri in zadnja dva znaka? Takole:

```
>>> b[3:-2]
'jam'
```

Že tale primer pokaže, zakaj je štetje od 0 in čudno pravilo, po katerem je prvi element rezine vključen, zadnji pa ne, tako smiselno in uporabno.

Vendar še nismo končali. Izpustimo lahko tudi zgornjo in spodnjo mejo. Kaj dobimo v tem primeru? Cel niz. Je to uporabno? Na nizih pravzaprav ne. Pri čem drugem pa nam bo prišlo še prav.

Poleg meja rezine lahko podamo tudi *korak*. Namesto vsakega znaka lahko zahtevamo, recimo, vsak drugi znak, tako da dodamo še eno dvopičje, ki mu sledi velikost koraka.

```
>>> '0123456789'[2:9]
'2345678'
>>> '0123456789'[2:9:2]
'2468'
>>> '0123456789'[2:9:3]
'258'
>>>
'0123456789'[2:9:4]
'26'
```

Korak je lahko, tako kot pri `range` tudi negativen. V tem primeru je potrebno zamenjati meji - prva mora biti višja od druge.

```
>>> '0123456789'[9:2:-1]
'9876543'
>>> '0123456789'[9:2:-2]
'9753'
```

Meje smemo seveda spet tudi izpuščati:

```
>>> '0123456789'[9::-1]
'9876543210'
```

Takole obrnemo predavatelja.

```
>>> 'demšar janez'[::-1]
'zenaj rašmed'
```

Vse tole je videti nekoliko zapleteno in tuje. In morda je: povaditi bo treba, pa se bo udomačilo. Pa se splača? Za odbijanje znakov od nizov? Se! Lepota

je v tem, da se natanko enako kot indeksiranje in rezanje nizov obnaša tudi indeksiranje seznamov in vsega drugega. Pa še kaj - zanke `for`, recimo, ki jih bomo vsak čas spoznali.

Poglejmo si torej še rezanje seznamov.

```
>>> imena = ["Anze", "Benjamin", "Cilka", "Dani", "Eva", "Franc"]
>>> imena[2:5]
['Cilka', 'Dani', 'Eva']
>>> imena[2:]
['Cilka', 'Dani', 'Eva', 'Franc']
>>> imena[:2]
['Anze', 'Benjamin']
>>> imena[:-2]
['Anze', 'Benjamin', 'Cilka', 'Dani']
>>> imena[-2:]
['Eva', 'Franc']
>>> imena[::-1]
['Franc', 'Eva', 'Dani', 'Cilka', 'Benjamin', 'Anze']
```

Že videno. Rezanje seznamov se vede enako kot rezanje nizov. Rezanje terk pa prav tako.

Spreminjanje seznamov z indeksiranjem in rezanjem

Tole je preprosto. Vsak element seznama se vede na nek način kot spremenljivka: lahko mu priredimo vrednost in s tem "povozimo" prejšnjo vrednost.

```
>>> imena
['Anze', 'Benjamin', 'Cilka', 'Dani', 'Eva', 'Franc']
>>> imena[3] = "Daniel"
>>> imena ['Anze', 'Benjamin', 'Cilka', 'Daniel', 'Eva', 'Franc']
```

Tako kot prej lahko tudi zdaj indeksiramo od spredaj ali od zadaj.

Pa rezine? Glede na to, da je rezina podseznam, moramo tudi pri prirejanju rezin prirejati podsezname.

```
>>> imena
['Anze', 'Benjamin', 'Cilka', 'Daniel', 'Eva', 'Franc']
>>> imena[1:4] = ["Ben", "Cecilija", "Dani"]
>>> imena
['Anze', 'Ben', 'Cecilija', 'Dani', 'Eva', 'Franc']
```

Mora biti seznam, ki ga prirejamo, enako dolg kot rezina, ki jo bomo povozili? Ne, čemu? Takole zamenjamo tri z dvema:

```
>>> imena[1:4] = ["Ben-Cecil", "Daniel"]
>>> imena
['Anze', 'Ben-Cecil', 'Daniel', 'Eva', 'Franc']
```

Ali pa nobenega s tremi:

```
>>> imena[2:2] = ["D2", "D3", "D4"]
>>> imena
['Anze', 'Ben-Cecil', 'D2', 'D3', 'D4', 'Daniel', 'Eva', 'Franc']
```

Z rezinami lahko tudi pobrišemo del seznama.

```
>>> imena
['Anze', 'Ben-Cecil', 'D2', 'D3', 'D4', 'Daniel', 'Eva', 'Franc']
>>> imena[2:5]
['D2', 'D3', 'D4']
>>> imena[2:5] = []
>>> imena
['Anze', 'Ben-Cecil', 'Daniel', 'Eva', 'Franc']
```

Za brisanje obstaja še veliko drugih načinov, recimo tale

```
>>> imena
['Anze', 'Ben-Cecil', 'Daniel', 'Eva', 'Franc']
>>> del imena[2]
>>> imena
['Anze', 'Ben-Cecil', 'Eva', 'Franc']
>>> del imena[1:3]
>>> imena
['Anze', 'Franc']
```

Če kdo pričakuje, da bomo zdaj povedali še, da enako delamo tudi z nizi in terkami ... se moti. Nizov in terk ne moremo spreminjati!

```
>>> b = 'Benjamin'
>>> b[2]='a'
Traceback (most recent call last):
File "", line 1, in TypeError: 'str' object does not support item assignment
```

Tu je torej osnovna razlika med seznamom in terko: seznam je spremenljiv, terka ne. Tudi niza ne moremo spreminjati, kakor smo pravkar videli. Kako pa bi potem zamenjali tretji znak niza b s črko 'a'?

```
>>> b = b[:2] + 'a' + b[3:]
>>> b
'Beajamin'
```

To je seveda nerodno. Čemu je torej tako? Čemu ne moremo spreminjati nizov tako, kot sezname? (In čemu sploh ta trapasta terka?) Videli bomo, da nam pride včasih zelo zelo prav, da so nekateri objekti nespremenljivi. (Nekateri - pogosto tudi jaz - pravijo celo, da jeziki sploh ne bi smeli dopuščati spreminjanja spremenljivk.) Nize pa si v resnici redko želimo spreminjati - da, pogosto jih bomo sestavljali ali obtesovali, zelo redko pa si želimo spreminjati posamezne črke. Zato nas to, da so konstantni, ne bo preveč motilo, velikokrat pa nam bo koristilo.

Računske operacije na seznamih (in drugod)

V svojem prvem soočenju s programiranjem smo spoznali aritmetične izraze: seštevali in množili smo števila, jih kvadrirali in računali sinuse. Zadnjič smo naleteli na logične izraze, kjer smo računali z logičnimi vrednostmi, `True` in `False`. Danes je čas, da se nehamo čuditi ob vsakem izrazu posebej in jim dajati pridevke "aritmetični" "logični" in tako naprej. Računati se da pač z različnimi rečmi in ena od teh reči so tudi seznami.

Lahko sezname seštevamo? Kaj dobimo, če seštejemo `[2, 5, -1]` in `[3, 7, 4]`? Dobimo `[5, 12, 3]` ali `[2, 5, -1, 3, 7, 4]`? Oboje bi bilo smiselno, odgovor pa je takšen: če so se seznami doslej vedli tako podobno nizom, naj se še glede seštevanja.Ker je `'abc' + 'def'` enako `'abcdef'`, naj bo tudi `[2, 5, -1] + [3, 7, 4]` enako `[2, 5, -1, 3, 7, 4]`.

Podobno je z množenjem. `[2, 5, -1] * 2` bi moralo biti po vsej logiki enako `[2, 5, -1] + [2, 5, -1]`, se pravi `[2, 5, -1, 2, 5, -1]`. In tudi je.

K seznamu lahko prištejemo le seznam, k nizu niz, k terki terko. Vse tri pa lahko pomnožimo s celim številom - in z ničemer drugim.

Poleg `+` in `*` pa poznajo vsi trije še dva operatorja, `in` in `not in`. S prvim vprašamo, ali seznam oz. terka vsebujeta določen element in ali niz vsebuje določen podniz.

```
>>> 1 in [1, 2, 3]
True
>>> 4 in [1, 2, 3]
False
>>> 'in' in 'Benjamin'
True
>>> 'an' in 'Benjamin'
```

Drugi je seveda ravno nasproten, z njim se vprašamo ali seznam (niz) *ne vsebuje* elementa (podniza).

```
>>> 1 not in [1, 2, 3]
False
>>> 4 not in [1, 2, 3]
True
>>> 'in' not in 'Benjamin'
False
>>> 'an' not in 'Benjamin'
True
```

V resnici `not in` ni prav potreben, saj je `x not in l` isto kot `not x in l`.